CS134: Searching
Announcements & Logistics

• **Lab 8** returned!

• **Lab 9 part 1 feedback returned**: let us know if you have any questions!

• **Lab 9 Boggle**
  - **Completed version of all classes** due next Wed/Thur
  - Make sure you thoroughly test your code

---

**Do You Have Any Questions?**
Last Time: Iterators

• Learned about **iterables** and **iterators**

• An object is considered **iterable** if it supports the `iter()` function (special method `__iter__` is defined): e.g, lists, strings, tuples

  • When an **iterable** is passed to the `iter()` function, it creates and returns an **iterator**

  • An **iterator** object can generate values **on demand**

    • **Supports the** `next()` function (and `__next__` method) which simply provides the "next" value in the sequence
Today and Next Week

• Briefly introduce how we measure efficiency in Computer Science
• Analyze the efficiency of some of our algorithms and data structures

Next Monday:
• Evaluate sorting algorithms and their efficiency

Last 5 classes: Introduction to Java (and Python review)
• Computational thinking and logic stays the same across programming languages
• We will focus on how the two languages are different in their syntax and structure
Measuring Efficiency
Measuring Efficiency

- How do we measure the efficiency of our program?
  - We want programs that run "fast"
  - How do we measure?
- One idea: use a stopwatch to see how long it takes
  - Is this a good method?
  - What is the stopwatch really measuring?
  - How long does this piece of code take on this machine on this particular input
- Machine (and input) dependent
  - We want to evaluate our program’s efficiency, not the machine's speed
- Cannot make any general conclusions using a stopwatch
  - Might not tell us how fast the program runs on different inputs/machines
Efficiency Metric: Goals

We want a method to evaluate efficiency that:

- **Is machine and language independent**
  - Analyze the *algorithm* (problem-solving approach)

- **Provides guarantees that hold for different types of inputs**
  - Some inputs may be "easy" to work with while others are not

- **Captures the dependence on input size**
  - Determine how the performance "scales" when the input gets bigger

- **Captures the right level of specificity**
  - We don't want to be too specific (cumbersome)
  - Measure things that matter, ignore what doesn't
Platform/Language Independent

**Machine and language independence**

- We want to evaluate how good the algorithm is, rather than how good the machine or implementation is
- Basic idea: Count the **number of steps** taken by the algorithm
- Sometimes referred to as the "running time"
Worst-Case Analysis

- We can't just analyze our algorithm on a few inputs and declare victory

- **Best case.** Minimum number of steps taken over all possible inputs of a given size

- **Average case.** Average number of steps taken over all possible inputs of a given size

- **Worst case.** Maximum number of steps taken over all possible inputs of a given size.

- Benefit of worst case analysis:

  - Regardless of input size, we can conclude that the algorithm always does *at least as well as* the pessimistic analysis.
Dependence on Input Size

- We generally don't care about performance on "small inputs"
- Instead we care about "the rate at which the completion time grows" with respect to the input size
- For example, consider the area of a square or circle: while the formula for each is different, they both grow at the same rate wrt radius
  - Doubling radius increases area by 4x, tripling increases by 9x
Dependence on Input Size: Big-O

- Big-O notation in Computer Science is a way of quantifying (in fact, upper bounding) the growth rate of algorithms/functions with respect to input size.

- For example:
  - A square of side length $r$ has area $O(r^2)$.
  - A circle of radius $r$ has area $O(r^2)$.

Doubling $r$ increases area 4×. Tripling $r$ increases area 9×.
Dependence on Input Size: Big-O

- Big-O notation captures the rate at which the number of steps taken by the algorithm grows wrt size of input $n$, "as $n$ gets large"

- Not precise by design, it ignores information about:
  - Constants (that do not depend on input size $n$), e.g. $100n = O(n)$
  - Lower-order terms: terms that contribute to the growth but are not dominant: $O(n^2 + n + 10) = O(n^2)$

- Powerful tool for predicting performance behavior: focuses on what matters, ignores the rest

- Separates fundamental improvements from smaller optimizations

- We won't study this notion formally: covered in CS136 and CS256!
Understanding Big-O

- Notation: \( n \) often denotes the number of elements (size)

- **Constant time** or \( O(1) \): when an operation does not depend on the number of elements, e.g.
  - Addition/subtraction/multiplication of two values, or defining a variable etc are all constant time

- **Linear time** or \( O(n) \): when an operation requires time proportional to the number of elements, e.g.:
  ```python
  for item in seq:
    <do something>
  ```

- **Quadratic time** or \( O(n^2) \): nested loops are often quadratic, e.g.,
  ```python
  for i in range(n):
    for j in range(n):
      <do something>
  ```
Big-O: Common Functions

- Notation: \( n \) often denotes the number of elements (size)
- Our goal: understand efficiency of some algorithms at a high level
Lists vs. Linked Lists

Efficiency Trade Offs
Lists vs Linked Lists

- **Linked Lists**: “pointer-based” data structure, items need not be contiguous in memory

- **Lists**: index-based data structure (sometimes called *arrays*), items are always stored contiguously in memory
Lists vs Linked Lists

- **Linked Lists:** Can grow and shrink on the fly: do not need to know size at the time of creation (therefore no wasted space!)

```
  _value   _value   ...   None
  5        3        11
  _rest    _rest    ...
```

- **Lists:** Need to know size (or use some default value) at the time of creation, can waste space by leaving room for future insertions

```
  head
  5  3  11 ...
  0  1  2 ...
```
An Aside: What exactly is Python's list?

- It's complicated: Python's list implementation is a hybrid
- For today's lecture, we will assume it's an array-based structure (lower picture)
Array vs Linked Lists

- Inserts at the beginning: which one is better?
Array vs Linked Lists

- Linked list steps:
  - Point head to new element
  - Point rest of new element to old list
  - These steps don't depend on size of list
  - Therefore, run-time is **constant**, that is, $O(1)$ time
Array vs Linked Lists

• Now consider an array-based list

• To insert at index 0, we need to shift every element over by one spot
  • This takes time proportional to the size: linear time or $O(n)$

• So when are arrays more efficient?
  • When **indexing** elements: they give **direct access** $O(1)$
  • Linked list: we need to traverse the list to get to the element $O(n)$
So Which is Better?

• It depends!

• **Time-space tradeoff**: try to find a balance between *time efficiency* and *space efficiency*

• Think about what list operations are required the most for your program

• Choose accordingly
Searching in an Array
Searching in an Array

- Let us discuss how quickly we can search for an item in an array-based list.

```python
def linearSearch(val, myList):
    for elem in myList:
        if elem == val:
            return True
    return False
```

- Might return early if `val` is the first item in `myList`, but we are interested in the **worst case analysis**; this happens if `val` is not in `myList` at all.

```
8  5  3  11  ...  
0  1  2  3
```
Searching in an Array

• In the worst case, we have to walk through the entire sequence

• Takes linear time, or \( O(n) \)

```python
def linearSearch(val, myList):
    for elem in myList:
        if elem == val:
            return True
    return False
```

Might return early if \( val \) is first item in \( myList \), but we are interested in the **worst case analysis**; this happens if \( val \) is not in \( myList \) at all
Searching in an Array

- Can we do better?
  - Not if the elements are in arbitrary order
- What if the sequence is sorted?
  - Can we utilize this somehow and search more efficiently?

How do we search for an item (say 10) in a sorted array?
Example: Dictionary

• How do we look up a word in a (physical) dictionary?
• Words are listed in alphabetical order
Example: Dictionary

Finding the definition of "octopus"

Open pages at ~half, is word on left or right?

Open pages at ~half, is word on left or right?

Open pages at ~half, is word on left or right?

Open pages at ~half, is word on left or right?

Find the word!
How Good is This Method?

- **Goal:** Analyze # pages we need to look at until we find the word

- We want the worst case: possible to get lucky and find the word right on the middle page, but we don't want to consider luck!

- Each time we look at the “middle” of the remaining pages, the number of pages we need to look at is divided by 2

- A 1024-page dictionary requires at most 11 lookups: 1024 pages, < 512, <256, <128, <64, <32, <16, <8, <4, <2, <1 page.

- Only needed to look at 11 pages out of 1024!

- Challenge: What if we have an \( n \) page dictionary, what function of \( n \) characterizes the (worst-case) number of lookups?
Logarithms: our favorite function

- Logarithms are the inverse function to exponentiation
- \( \log_2 n \) describes the exponent to which 2 must be raised to produce \( n \)
- That is, \( 2^{\log_2 n} = n \)
- Alternatively:
  - \( \log_2 n \) (essentially) describes the number of times \( n \) must be divided by 2 to reduce it to below 1
- For us, here’s the important takeaway:
  - How many times can we divide \( n \) by 2 until we get down to 1
  - \( \approx \log_2 n \)
Binary Search

- The recursive search algorithm we described to search in a sorted array is called binary search.
- It is much, much more efficient than a linear search: $O(\log n)$ time.
  - **Note:** $\log n$ grows much more slowly compared to $n$ as $n$ gets large.
- Let's implement this technique.

```python
def binarySearch(aList, item):
    """Assume aList is sorted.
    If item is in aList, return True;
    else return False.""
    pass
```
Binary Search

- Base cases? When are we done?
  - If list is too small (or empty)
  - If item is the middle element

```python
def binarySearch(aList, item):
    """Assume aList is sorted.
    If item is in aList, return True;
    else return False.""
    n = len(aList)
    mid = n // 2
    # base case 1
    if n == 0:
        return False

    # base case 2
    elif item == aList[mid]:
        return True
```

mid = n//2
Binary Search

- Recursive case:
  - Recurse on left side if item is smaller than middle
  - Recurse on right side if item is larger than middle

If item < aList[mid], then need to search in aList[:mid]

mid = n//2
Binary Search

• Recursive case:
  • Recurse on left side if item is smaller than middle
  • Recurse on right side if item is larger than middle

If item > aList[mid], then need to search in aList[mid+1:]
def binarySearch(aList, item):
    """Assume aList is sorted. If item is in aList, return True; else return False."""
    n = len(aList)
    mid = n // 2
    # base case 1
    if n == 0:
        return False

    # base case 2
    elif item == aList[mid]:
        return True

    # recurse on left
    elif item < aList[mid]:
        return binarySearch(aList[:mid], item)

    # recurse on right
    else:
        return binarySearch(aList[mid + 1:], item)
The end!